На предыдущем занятии мы с вами определили базовый класс GeomBase и производный от него класс Line:

```language-cpp
class GeomBase {
protected:
    int x0{0}, y0{0}, x1{0}, y1{0};
public:
    void set_coords(int x0, int y0, int x1, int y1)
    {
        this->x0 = x0; this->y0 = y0;
        this->x1 = x1; this->y1 = y1;
    }

    virtual void draw() const
    {
        printf("GeomBase: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};


class Line : public GeomBase {
private:
    double length{0.0};
public:
    virtual void draw() const
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

Как правило, виртуальные методы объявляются с целью их переопределения в дочерних классах (иначе виртуальность теряет смысл). При этом сигнатура переопределяемого метода должна четко соблюдаться. Например, если в дочернем классе Line виртуальный метод draw переопределить, добавив один параметр:

```language-cpp
class Line : public GeomBase {
private:
    double length{0.0};
public:
    virtual void draw(bool fl_draw=true) const
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
```

```
    }
};
```

А, затем, в функции main создать объект этого класса и вызвать метод draw:

```language-cpp
int main()
{
    Line* ptr_ln = new Line;

    ptr_ln->draw();

    delete ptr_ln;
    return 0;
}
```

То программа скомпилируется без каких-либо ошибок и замечаний. Однако виртуальный метод draw базового класса GeomBase при этом не был переопределен. Если мы сформируем указатель ptr_b типа GeomBase и через него вызовем метод draw:

```language-cpp
int main()
{
    Line* ptr_ln = new Line;
    GeomBase* ptr_b = ptr_ln;

    ptr_ln->draw();
    ptr_b->draw();

    delete ptr_ln;
    return 0;
}
```

то вызовется метод базового класса, в консоли увидим строчки:

```
Line: 0, 0, 0, 0
GeomBase: 0, 0, 0, 0
```

И все из-за наличия параметра у метода draw дочернего класса Line. Если этот параметр убрать, то в обоих случаях будет вызван метод дочернего класса:

```
Line: 0, 0, 0, 0
Line: 0, 0, 0, 0
```

Видите, как легко совершить ошибку, случайно прописав лишний параметр при переопределении виртуального метода. При этом компилятор никак на это не реагирует. Похожая ситуация может возникнуть, если в базовом классе меняется сигнатура виртуального метода. Из-за этого его переопределение в дочерних классах перестает существовать. Чтобы в процессе разработки программы исключить подобные ошибки переопределяемый виртуальный метод следует явно помечать ключевым словом override:

```language-cpp
class Line : public GeomBase {
private:
    double length{0.0};
public:
    virtual void draw() const override
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

Тогда при совпадении сигнатур программа скомпилируется без ошибок. **А если сигнатура изменится, например, добавим прежний параметр**:

```cpp
virtual void draw(bool fl_draw=true) const override
{
printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
}
```

```
**то компилятор сообщит об ошибке**. Поэтому все переопределяемые виртуальные методы следует помечать в дочерних классах ключевым словом override. И это касается именно виртуальных методов. С обычными таких проблем не возникает, т.к. они всегда вызываются каждый из своего класса (либо берутся из базового при отсутствии переопределения).

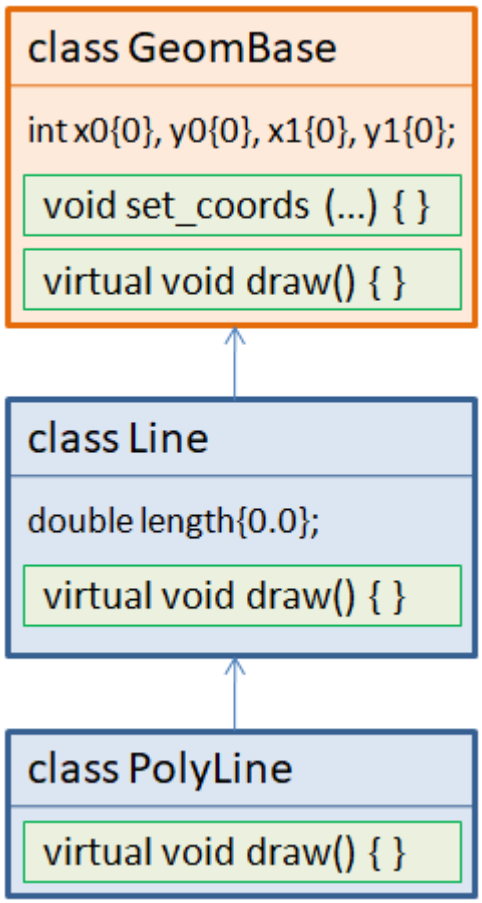
## Ключевое слово final

Давайте теперь представим, что мы объявляем еще один дочерний класс от класса Line:

```cpp
class PolyLine : public Line {
```
```

```
public:
    virtual void draw() const override
    {
        printf("PolyLine: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

Так тоже можно делать. В качестве базового, в общем случае, может выступать любой класс. В результате, получается следующая цепочка наследования:



Очевидно, если создается объект класса PolyLine, то метод draw будет вызываться из этого класса, как бы мы это не делали:

language-cpp
```
int main()
{
    Line* ptr_ln = new Line;
    PolyLine* poly = new PolyLine;

    GeomBase* ptr_b = ptr_ln;
    GeomBase* ptr_b_poly = poly;

    ptr_b->draw();
```

```
    ptr_b_poly->draw();

    delete ptr_ln;
    delete poly;
    return 0;
}
```

После запуска программы в консоли увидим:

```
Line: 0, 0, 0, 0
PolyLine: 0, 0, 0, 0
```

Но, что если по каким-либо причинам, мы бы захотели запретить дальнейшее переопределение метода draw? То есть, оставить только одну реализацию в классе Line. Для этого в конце прототипа метода следует прописать ключевое слово final следующим образом:

```language-cpp
class Line : public GeomBase {
...
    virtual void draw() const override final
    {
        printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1);
    }
};
```

После этого переопределение draw в классе PolyLine приведет к ошибке на этапе компиляции программы. И программисту ничего не останется, как убрать определение этого метода из класса.

Обратите внимание, что ключевое слово final можно применить только к виртуальным методам. С обычными методами оно не работает.

Виртуальные деструкторы

В заключение этого занятия рассмотрим еще один важный момент, связанный с наследованием классов. Давайте представим, что в программе создается объект дочернего класса Line следующим образом:

```language-cpp
int main()
{
    GeomBase* ptr_ln = new Line;
    ptr_ln->draw();
    delete ptr_ln;
```

```
    return 0;
}
```

На первый взгляд здесь все кажется корректным. Добавим теперь в классы GeomBase и Line деструкторы следующим образом:

```language-cpp
class GeomBase {
...
    ~GeomBase() { puts("Delete: GeomBase"); }
};


class Line : public GeomBase {
...
    ~Line() { puts("Delete: Line"); }
};
```

После запуска программы в консоли увидим:

```
Line: 0, 0, 0, 0
Delete: GeomBase
```

То есть, был вызван только один деструктор базового класса GeomBase. Очевидно, это произошло по той причине, что мы работаем с объектом через указатель на базовый класс. **Как же нам тогда поправить описание классов, чтобы вызывались оба деструктора: и базового и дочернего классов? Здесь нам на помощь приходят виртуальные методы. Деструкторы классов разрешено тоже делать виртуальными**. И, если пометить деструктор базового класса, как виртуальный:

```language-cpp
class GeomBase {
...
    virtual ~GeomBase() { puts("Delete: GeomBase"); }
};
```

то вместо него будет вызван деструктор дочернего класса Line – того объекта, который уничтожается. А деструктор дочернего класса, как мы уже знаем, автоматически вызывает и деструктор базового. В результате, при выполнении программы, в консоли увидим:

```
Line: 0, 0, 0, 0
Delete: Line
Delete: GeomBase
```

Вас может удивить, что деструктор, имеющий имя ~GeomBase был переопределен деструктором с именем ~Line(). Да, это особенность именно деструкторов. Их сигнатуры, на самом деле, в разных классах полностью совпадают на уровне таблиц виртуальных методов. Поэтому виртуальный деструктор базового класса успешно переопределяется деструктором любого дочернего класса. И если мы знаем, что класс предполагается использовать при наследовании, то его деструктор лучше сразу помечать, как виртуальный, даже если он ничего не делает. Иначе мы рискуем не вызвать деструктор дочернего класса, а это уже может быть критично. **Вообще, если класс имеет хотя бы один виртуальный метод, то его деструктор следует делать виртуальным**. Некоторые компиляторы даже выдают предупреждение, если этого не сделать. Этот факт еще раз подчеркивает важность объявления виртуальных деструкторов в классах, которые участвуют в наследовании.